Breaking Type Safety in Go: An Empirical Study on the Usage of the `unsafe` Package

Diego Elias Costa, Suhaib Mujahid, Rabe Abdalkareem, and Emad Shihab *Senior Member, IEEE*

Abstract—A decade after its first release, the Go language has become a major programming language in the development landscape. While praised for its clean syntax and C-like performance, Go also contains a strong static type-system that prevents arbitrary type casting and memory access, making the language type-safe by design. However, to give developers the possibility of implementing low-level code, Go ships with a special package called `unsafe` that offers developers a way around the type safety of Go programs. The package gives greater flexibility to developers but comes at a higher risk of runtime errors, chances of non-portability, and the loss of compatibility guarantees for future versions of Go.

In this paper, we present the first large-scale study on the usage of the `unsafe` package in 2,438 popular Go projects. Our investigation shows that `unsafe` is used in 24% of Go projects, motivated primarily by communicating with operating systems and C code, but is also commonly used as a means of performance optimization. Developers are willing to use `unsafe` to break language specifications (e.g., string immutability) for better performance and 6% of the analyzed projects that use `unsafe` perform risky pointer conversions that can lead to program crashes and unexpected behavior. Furthermore, we report a series of real issues faced by projects that use `unsafe`, from crashing errors and non-deterministic behavior to having their deployment restricted from certain popular environments. Our findings can be used to understand how and why developers break type safety in Go, and help motivate further tools and language development that could make the usage of `unsafe` in Go even safer.

Index Terms—Go language, `unsafe`, type safety, software packages, Empirical Study.

✦

1 INTRODUCTION

The famous Uncle Ben's quote "With great power comes great responsibility" is a proverb that can be aptly applied to unsafe packages or libraries in programming languages. Statically-typed programming languages, such as Java, Rust, and Go restrict developers' freedom (power) by requiring every variable and function to have a type known at compile-time, in favor of a safer environment, capturing illegal type conversions and memory accesses during compilation. While efficient at identifying type violations, the restriction to write type-safe implementations would make it nearly impossible to write low-level code, such as system calls and pointer operations. Language designers address this problem by including a backdoor to violate the type system in the form of unsafe packages. Java has `sun.misc.Unsafe`, Rust has `unsafe Rust`, and Go has the `unsafe` package. Unsafe packages give the much needed flexibility to write type-unsafe functions, for low-level implementations and optimizations, but need to be used with extreme care by developers [39], [53].

Go is a statically-typed and compiled programming language released over a decade ago by engineers at Google [3]. Its simple syntax and high efficiency has made Go one of the major programming languages in the current development landscape [6], [45]. Go has a strong static type system, but

- *D. E. Costa, S. Mujahid, and E. Shihab are with the Data-driven Analysis of Software (DAS) Lab at the Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada.*
  E-mail: diego.costa, suhaib.mujahid, emad.shihab@concordia.ca
- *R. Abdalkareem is with Software Analysis and Intelligence Lab (SAIL), School of Computing, Queen's University, Canada.*
  E-mail: abdrabe@gmail.com

ships with the `unsafe` package [39] to offer developers the possibility of implementing low-level code. This package offers a way-around the type safety of Go programs, but it comes with a series of important drawbacks. First, programs that use `unsafe` may not be portable to different CPU architectures, and are not guaranteed to be compatible to future versions of Go [2]. Second, some `unsafe` operations contain hidden complexities that can cause the program to exhibit non-deterministic errors. For instance, Go contains pointers and a Garbage Collector (GC), hence, manipulating (unsafe) pointers without proper care may cause the GC to release unwanted variables [10].

Given that the benefits of writing type-safe code are well-known and lifting this safety net puts programs at a higher risk of runtime errors [53], why do developers break type safety in Go? There is no shortage of articles in the web warning against the perils of using `unsafe` [5], [13], [39], and maintainers of Go have had extensive debates over the need and consequences of keeping this package in the language [23]. However, it is hard to derive effective measures on how to handle the risks and benefits of `unsafe` package without knowing the extent in which developers use it, what they use it for, and what are the real risks of breaking type safety in Go projects. Our study is a step towards acquiring this understanding.

In this paper, we perform a mix-method empirical study involving 2,438 popular Go open-source projects. We first develop a parser to identify usages of the `unsafe` package throughout the development history of Go projects. Then, we perform a manual analysis to evaluate the prevalence of `unsafe` usage and their consequences in the most popular Go projects. Our study focuses on answering the following research questions:

RQ1: (Prevalence of `unsafe`) Is `unsafe` widely used in open-source Go projects? We found that 24% of the studied Go projects use `unsafe` at least once in their code-base. The usage of `unsafe` grows with the project size, with projects using `unsafe` on 17% of their packages. The package `unsafe` is used in a wide variety of project domains (e.g., networking, blockchains, containers, development tools, databases). We also found that projects that rely most heavily on the `unsafe` package, with more than 100 call-sites, are the ones that implement bindings to other platforms and/or programming languages.

RQ2: (Prevalence of `unsafe`) What is `unsafe` used for? We catalogued six groups of usage-patterns related to `unsafe` in Go. The majority of `unsafe` usages were motivated by integration with operating systems and C code (45.7%), but developers also frequently use `unsafe` to write more efficient Go code (23.6%). Less frequently, developers use `unsafe` to perform atomic operations, dereferrence values through reflection, manipulate memory addresses and get the size of objects in memory.

RQ3: (Consequences of `unsafe`) What are the risks of using `unsafe`? At least 6.6% of the investigated projects that use `unsafe` have invalid pointer conversions, a risky usage of the API that may cause crashing bugs and lead programs to unexpected runtime behavior. Developers report a variety of issues caused by using `unsafe`, from having their deployment restricted by environment platforms, to crashing and non-deterministic errors found in production.

Our study provides empirical evidence that contributes towards a safer Go programming language. Our findings show that the usage of `unsafe` is widespread, covering all sorts of project domains, and is motivated by integration with Operating Systems, C programs, and more efficient implementations. The usage patterns we catalogued can be used by tool designers to better assist developers when performing unsafe operations, as well as guiding standard packages to improve documentation to cover the most common use-cases. Furthermore, our results indicate that even popular projects are not immune to well-known pitfalls associated with the package usage and that projects that use `unsafe` report a variety of unsafe-related issues. Our results also help developers at identifying potential risks and pitfalls to avoid when using the `unsafe` package.

This paper is organized as follows: Section 2 introduces the `unsafe` package and the concepts that we use throughout the paper. Section 3 presents our methodology used for our study. The results of our study are presented in three sections, Section 4, Section 5, and Section 6 presents the results of RQ1, RQ2, and RQ3, respectively. We discuss our findings and implications in Section 7. Then, in Section 8 we present the related work and discuss the threats to the validity of our results in Section 9. Finally, we conclude our study in Section 10.

2 BACKGROUND

In this section, we introduce the `unsafe` package API and illustrate some of its use-cases. We describe the risks of using `unsafe` and dive into a unsafe-related pitfall that may cause programs to crash or exhibit a non-deterministic behavior.

2.1 Type safety in Go and the `unsafe` package

Go is a statically typed language. That is, the type of variables in a Go program are known at compile time, enabling the compiler to verify type inconsistencies and incompatible type conversions. A compiled Go program is guaranteed to be type-safe which prevents a myriad of runtime issues, unless developers use the `unsafe` package. As described in the official documentation shown in Figure 1, the `unsafe` package offers a step around the type safety of Go programs [39]. The `unsafe` package is quite compact, containing only three functions and one Pointer type in its API, shown in the Index of Figure 1-A. By using the `unsafe` package, developers have the flexibility needed to develop low-level code, such as full control over pointers (C-style), the ability to read and write arbitrary memory, access to object sizes, and the possibility to convert between any two types with compatible memory layouts (shown in Figure 1-B). Compared to unsafe libraries from other programming languages like the `sun.misc.Unsafe` from Java and `unsafe Rust`, the Go `unsafe` package is still much more restricted. By using their respective unsafe APIs, in addition to breaking type safety, developers can violate method contracts in Java [53] or use inline assembly functions in Rust [26].

2.2 Risks of using the `unsafe` package

In practical terms, breaking type safety means lifting the safety net that a compiler provides for developers in exchange for full-control of reading and writing memory. This by itself, puts developers at a higher risk of making mistakes that will ripple through to the production environment. For instance, converting between incompatible types may cause the program to eventually crash or misinterpret the memory layout of a variable causing unexpected program behavior. Aside from this, the `unsafe` package comes with a series of drawbacks described in its documentation [39], that need to be taken into careful consideration by developers:

Non-Portability: A regular package in Go can be compiled to different CPU architectures and operating systems without any changes in the code. Projects that use `unsafe`, however, may need to account for differences in the CPU architectures and it is up to developers to keep their packages portable. For instance, by traversing an array of integers using `unsafe`, developers need to account for the integer size which differs in x86 and x64 architectures. Another example is the reliance on system calls, which are accomplished by using both `syscall` with `unsafe` packages and are not often portable to different operating systems.

Not Protected by the Compatibility Guideline: Programs that are written in Go have the guarantee to work in future versions of the programing language, as long as they follow the compatibility guideline established in Go 1 [2]. However, using the `unsafe` package breaks this compatibility guarantee, as `unsafe` exposes the internal implementations of the language, which may change in future versions. Hence, programs that rely on `unsafe` are not guaranteed to work in future implementations of Go.
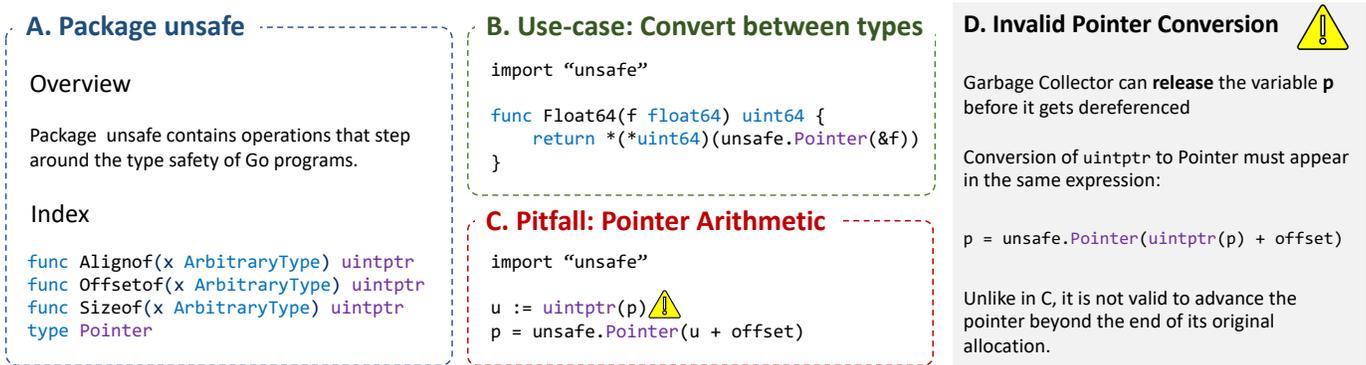
A. Package unsafe

Overview

Package unsafe contains operations that step around the type safety of Go programs.

Index

```
func Alignof(x ArbitraryType) uintptr
func Offsetof(x ArbitraryType) uintptr
func Sizeof(x ArbitraryType) uintptr
type Pointer
```

B. Use-case: Convert between types

```
import "unsafe"

func Float64(f float64) uint64 {
    return *(*uint64)(unsafe.Pointer(&f))
}
```

C. Pitfall: Pointer Arithmetic

```
import "unsafe"

u := uintptr(p) ⚠
p = unsafe.Pointer(u + offset)
```

D. Invalid Pointer Conversion ⚠

Garbage Collector can **release** the variable **p** before it gets dereferenced

Conversion of uintptr to Pointer must appear in the same expression:

```
p = unsafe.Pointer(uintptr(p) + offset)
```

Unlike in C, it is not valid to advance the pointer beyond the end of its original allocation.

Fig. 1: The unsafe package API (A), an example of how to use unsafe to convert between types in Go (B) and a pitfall related to performing pointer arithmetic in Go (C). On the right we explain the invalid pointer conversion issue and give an example of solution for a safe pointer arithmetic. All examples were taken from the official documentation [39]

2.3 Invalid Pointer Conversion

The unsafe package provides an extensive documentation on how to properly use the package and some of the pitfalls developers must avoid when breaking type safety [39]. A pitfall that is well-described in the package documentation is the invalid pointer conversion. As a rule of thumb, converting pointer addresses back to pointer variables is not valid in Go language. Pointer memory addresses in Go are of the type uintptr (unsigned integer pointer), which is a simple integer type, holding no reference or pointer-semantics. Since Go has a Garbage Collector that releases memory that is not referenced by any variable in the program, a variable uintptr holding the address of a pointer will not prevent the garbage collector from releasing the said pointer. Consequently, there is no guarantee that a memory address (type uintptr) contains a valid Go pointer variable to be dereferenced [10].

In some cases, however, there is a valid need to manipulate memory addresses and dereference them back to pointers. For instance, to perform pointer arithmetic operations (C-style) to traverse an array as shown in Figure 1-C. Developers need to be aware of the intricacies of manipulating low-level pointers in a language that contains a Garbage Collector, to prevent the Garbage Collector from releasing their variables in the middle of their operations. We show in Figure 1-D an example on how to properly handle pointer arithmetics in Go. In this case, developers should never store their uintptr into an intermediate variable (u), because at this point in the execution of a program the variable p can be released by the Garbage Collector.

This example shows that the pitfalls of handling unsafe are not always intuitive. To make matters worse, the issues that arise from using the invalid pointer conversion are non-deterministic and are unlikely to be encountered during software testing when memory pressure tends to be small.

3 METHODOLOGY

To understand the prevalence of unsafe and its impact in open source Go projects, our study has three main goals:

1) Understand the extent in which projects use unsafe package in their source-code (Section 4). To achieve this, we identify projects that use unsafe with a parser,

investigate whether the usage of the package changes as the project evolves and what are the categories of projects that more frequently rely on unsafe.
2) Understand what developers use unsafe for (Section 5). We investigate what are the most used features from the unsafe API and manually extract the most common usage patterns of unsafe in Go projects.
3) Understand the risks that using unsafe entails to projects (Section 6). We examine whether projects have invalid pointer conversions in their code and qualitatively evaluate issues related to the unsafe package.

In the following subsections, we describe the methodology used to achieve the goals of our study. We present the overview of our methodology in Figure 2.

3.1 Study dataset

To investigate the usage of unsafe on a large-set of popular Go projects, we first started by querying the GitHub REST API[1] to identify a representative set of open source Go project. To do so, we selected a set of the top 3,000 most starred Go repositories, as the number of stars is an indicator of the project popularity within GitHub repositories [14], [15]. Our dataset was collected on October 2nd, 2019.

Even within highly starred repositories, we may find inactive repositories or repositories not related to software development. To get a representative set of high-quality and active Go software development projects, we follow the methodology recommended by previous work [47] to further curate our dataset through the following criteria:

1) We filter out 89 archived projects, as these projects are no longer maintained by their developers. Archived projects are identified via a flag in the project's metadata.
2) We removed 371 inactive projects, by filtering out projects that have had no contribution 12 months prior to the data collection date (after October 2018).
3) We removed 22 projects with less than 10 commits in total, as these projects tend to be too young and are not representative of the projects we aim to investigate.
4) We also removed 56 educational repositories, which are fairly common on popular projects datasets. These projects are described as published books, programming
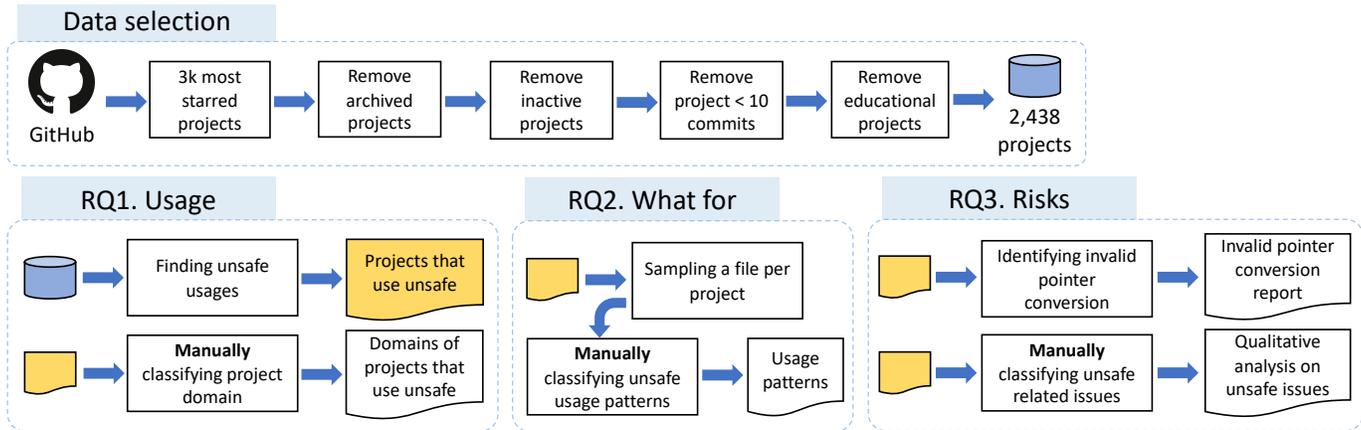
1. https://developer.github.com/v3/

Fig. 2: Overview of the methodology adopted in our study.

TABLE 1: Statistics of the projects in our dataset.

Statistics	Mean	Min	Median	Max
Age (months)	44.17	1	43	119
Stars	1,967.81	314	858	64,079
Forks	276.71	2	109	20,437
LOC	210,217.09	77	9,606	16,579,983
Commits	1,028.48	11	261	97,504

courses or any learning material, which are invaluable to the community, but are not representative of typical Go software. We removed these by manually inspecting project descriptions of our entire dataset.

5) Furthermore, 24 projects could not be cloned automatically by our scripts (e.g., invalid URL, project no longer available) and were removed from our study.

This process yields a dataset composed of 2,438 Go projects. Table 1 shows the summary statistic of the selected Go projects in our dataset. As shown in Table 1, our dataset contains very popular projects (median of 857 stars and 110 forks), with a relatively long development history (median of 3.5 years of development). The dataset contains eight projects with less than 150 LOC (minimum of 77 LOC), but in its large majority are composed of medium to large software projects (median of 9,6 KLOC). We do not focus on filtering projects based on their size, as we think the perils of breaking type safety affects projects of all sizes.

3.2 Identifying `unsafe` usages

To find whether projects make use of `unsafe` in their source-code, we build a parser for Go source code, using the support of Go's native `ast` package[2]. Our parser first analyzes the source code of each Go file (files with .go extension) in the selected projects and build an abstract syntax tree (AST) for each file. Then, the parser inspects the AST of Go files and identifies the function calls and type references to `unsafe`. First we run our parser at the latest snapshot of the projects obtained during our data collection. In addition, we also want to understand how the usage of `unsafe` changes as the project evolves. To that aim, we use our parser to analyze the snapshot of the first commit of each month in the projects' history, and For this analysis, we

2. https://golang.org/pkg/go/ast/

only consider the default branch of each project, identified via the GitHub API.

3.3 Classifying projects' domain

To complement the analysis on the usage of `unsafe`, we investigate the domain of the projects that use the `unsafe` package, as the domain has a direct influence on the need for breaking type safety. Intuitively, we expect projects that demand low-level implementation and optimizations, such as databases and file systems, to depend more on `unsafe` implementations than other type of projects such as data structure libraries and web applications. To identify the projects' domain, we manually inspect the description and documentation in their GitHub page and classify each project into a dominant domain, e.g., database, compiler, web server, development tool. The three-first authors classify the repositories using an open card-sort method [29], where labels are created during the labeling process and each new label is discussed among annotators and retroactively applied to previously classified projects. When different labels were assigned to the same project we discussed them to reach a consensus.

3.4 Classifying `unsafe` usage patterns

To investigate what developers aim to accomplish using `unsafe` in RQ2, we need to understand the most frequent usage patterns associated with the `unsafe` package. Each usage pattern may offer a rationale that will help us understand the proximate reason as to why developers opted for breaking type safety in their projects. To extract high-level usage patterns from the projects' source code, we resort to in-depth manual analysis of the code, documentation, and commit messages. It is important to notice that this analysis is very time consuming, as annotators need to recognize the context in which `unsafe` is being used, search for clues that indicate the reason behind the `unsafe` in commit messages and code documentation. Hence, we decide to perform such analysis on a statistically significant random sample of the projects that use `unsafe`. This sample is drafted to provide us with a representative set of the projects that rely on unsafe with 5% confidence interval at 95% confidence level.

We expect some projects to contain hundreds of `unsafe` call-sites while most projects may use `unsafe` sporadically

in their code, given the risks associated with the package. We want to identify patterns across projects and avoid biasing our analysis towards projects that rely more extensively on `unsafe` in their implementation. Hence, we perform a second sampling by randomly selecting a single file from each project to our analysis.

It is important to highlight that we opt to conduct this particular analysis at the file level as opposed to package-level for the following reasons: 1) Files are more fine-grained than packages and are expected to have a more cohesive structure where we could more easily derive the usage pattern; 2) We can analyze the context of usage of a single file, with support of `gitblame` to futher inspect commit messages, without the need to inspect method calls across different files, which would impose a prohibitive time cost to this analysis.

The first two authors independently labeled each file (one per project) using an open card-sort method [29]. Hence, similarly to the analysis of projects' domains, labels are created and assigned while inspecting the usage in source-code and git commits, and every new label is discussed among annotators and if necessary, retroactively applied to previously labelled usages. We assess the agreement of both annotators using the Cohen-Kappa metric [55] and both annotators discussed and merged the results.

3.5 Identifying invalid pointer conversions

In RQ3, we investigate the risks to projects caused by using the `unsafe` package. As discussed in Section 2, a major pitfall of using unsafe pointers is being unaware of issues related to invalid pointer conversions. This issue can lead to non-deterministic errors in programs, as the Garbage Collector may release unintended variables and causing the program to crash. We want to investigate whether invalid pointer conversions do occur in the most popular Go projects. To automatically identify suspicious cases of invalid pointer conversions, we resort to existing static analyzers, as this issue is well-documented and covered by a go code analyser: Vet [4]. Go Vet was created by maintainers of the Go language and is shipped natively with Golang. It identifies suspicious cases of invalid pointer conversion when run with the flag `unsafeptr`. Go Vet parses the code and applies a set of heuristics based on known ill-formed expressions (such as the one in Figure 1), exporting every suspicious case as a json file. As with any heuristic-based tool, Go Vet may miss cases that do not fall into the most known patterns of invalid pointer conversions.

3.6 Classifying **unsafe** related issues

Another method for understanding the risks associated with `unsafe` usage is to investigate the issues open in Github related with the package use. To analyze and classify the issues related to `unsafe`, we first mine all issues from the repository of all projects that use `unsafe`. In GitHub, "issue" is an umbrella term that encloses pull requests, bugs, questions to maintainers and requests for new features. We find candidate issues for our analysis by applying a keyword search for "unsafe" and two variations ("un-safe" and "not safe") in the issue title. The keyword approach is prone to false-positives, especially given that the keywords are used in several different contexts (e.g., multi-threading), hence, we need to filter out false-positives from

this candidate set. The first author manually inspected each issue and removed the false-positives. Then the first two authors proceed to analyze the issue title, body and related commit code to group issues based on their similarities. Similarly to the previously described methods, we resort to the open-card methodology [29] and evaluate the interrated agreement using the Cohen-Kappa interrater method [55]. We discuss the disagreement in a second round to reach a consensus in the classification. Note that the goal of this analysis is not to find all issues related to `unsafe`, but rather to classify a sample of possible unsafe-related issues to provide qualitative insights about the problem related to use the `unsafe` package.

3.7 Replication Package

To facilitate verification and advancement of research in the field, we provide a replication package containing the list of projects analyzed, the data extracted from the projects, and the scripts used to process and analyze our RQs [3].

4 IS UNSAFE WIDELY USED IN OPEN-SOURCE GO PROJECTS?

Motivation: We start the study by investigating to what extent developers use `unsafe` in open-source Go projects. This will help us understand how frequently developers abandon type safety guarantees to implement their programs; how `unsafe` usages change as the project evolves and which domain of projects make more use of `unsafe`. This understanding is crucial to motivate the development of better solutions related to `unsafe` usage in Go. We analyze this question under three complementary aspects:

1) **Usage:** How often does a Go project uses `unsafe` in their source-code? This will help us understand how frequently developers abandon type safety guarantees to implement their programs.

2) **Trend:** Does the usage of `unsafe` change over the evolution of a project? This analysis will give us insights on whether `unsafe` usage increases and spreads to multiple packages as the project evolves, or if developers make the conscious effort of isolating `unsafe` to mitigate its risks.

3) **Domain:** What domain of projects rely on `unsafe`? With this analysis, we aim at identifying what categories of projects are more susceptible to breaking type safety.

4.1 How often does a Go project use **unsafe**?

Approach: We run our parser to identify `unsafe` usages in every Go file present at the latest snapshot of projects in our dataset, but filter out usages identified in the source-code of the project dependencies. A commonly employed practice in Go projects for managing dependencies is to include all dependencies in the projects structure, in a folder called "vendor". Hence, we exclude all reports of `unsafe` usage originating from vendor folders.

Results: Table 2 shows that from the 2,438 evaluated Go projects, **592 (24%) make use of unsafe directly in the project source-code**. The extent in which projects use `unsafe` varies considerably. In Figure 3 we show the distribution of the `unsafe` usage in our dataset as a Kernel-density estimation plot, under three granularities: call-sites,

---

3. https://zenodo.org/record/3871931

(a) # of `unsafe` call-sites.

(b) # of files with `unsafe` call-sites.

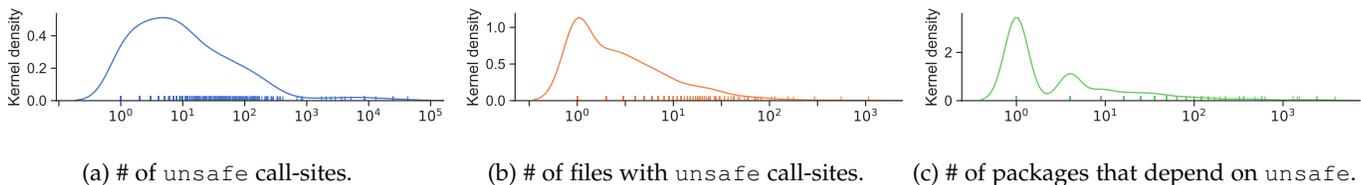(c) # of packages that depend on `unsafe`.

Fig. 3: Distribution of `unsafe` usage in three granularity levels (call-site, files and packages) per project as a KDE density estimation plot. Note that the x-axis is logarithmic.

TABLE 2: Statistics on the projects that use `unsafe`.

Statistics	#	%
All projects	2,438	100.00
Projects that use `unsafe`	592	24.28
Projects with > 100 `unsafe` call-sites	69	2.83

files, and packages. The call-sites plot accounts for every `unsafe` operation called in a project, the files indicate how many files depend on `unsafe` and the package plot shows how many modules in Go depend on `unsafe`. As evidenced by the Figure 3a peak, the majority of projects (57%) contains fewer than 10 calls to `unsafe` operations in their source-code. Consequently, most projects concentrate their `unsafe` calls in at most 2 files and a single package, keeping the `unsafe` usage well-localized in the code (see the distribution shown in Figures 3b and 3c). Yet, we found that 69 projects (2.8%) in our dataset rely heavily on `unsafe` in their project, with more than 100 call-sites present in their source-code.

## 4.2 Does the usage of `unsafe` change as projects evolve?

Approach: Since we want to analyze whether and how the usage of `unsafe` changed over the course of the project development, we examine the use of `unsafe` package at different snapshot of the projects history. As our dataset contains in median projects with almost 4 years of development (see Table 1), we focus on analyzing the trend of `unsafe` usage on the period between January 2015 to September 2019 (a month before our data collection time). In addition, to perform a sound analysis of the usage of `unsafe` over the years, we only conduct this analysis on projects that fulfill the following criteria: 1) projects that have at least 10 `unsafe` operation calls at the latest snapshot of their source code, as these will show a more meaningful evolution of `unsafe` usage; and 2) projects that were being actively developed during this entire period, i.e., projects with at least one commit pushed in each year. It is important to notice that we applied these criteria to avoid skewing our results towards a particular time frame. For instance, younger projects could skew our analysis as a higher number of projects would be accounted for in the most recent years.

Results: Figure 4 shows the evolution of the 270 projects in terms of number of packages and packages using `unsafe` (Figure 4a) together with the percentage of packages that rely on `unsafe` (Figure 4b). In both plots, the thick line represents the mean value and the colored area shows the 95% confidence interval of the data at each month.

As shown in Figure 4a, the number of packages using `unsafe` grows similarly to the growth of total number of packages in a project. As the number of packages increased
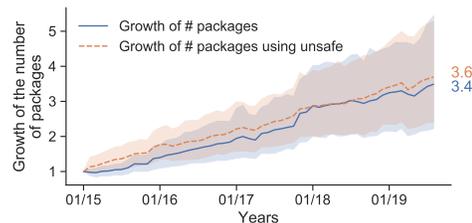
(a) Evolution of 270 projects compared to their state in January 2015 (baseline) in terms of the number of packages and number of packages using `unsafe`.
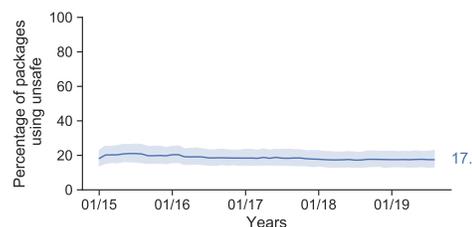
(b) Evolution of the percentage of packages that use `unsafe` in 270 projects.

Fig. 4: Analysis of projects' evolution in terms of number of packages and the share of packages that rely on `unsafe`.

by 3.5 times in our studied projects, the number of packages using unsafe also increased at a similar rate (3.7 times). This indicates that the usage of `unsafe` evolves linearly as the project size increases, in the evaluated projects. In addition, Figure 4b shows the proportion of packages using `unsafe`, which remains steadily around 17% as projects evolve. Our results paint a mixed picture; while the dependency of `unsafe` does not spread out to a larger proportion of packages, it has also not decreased as projects grow in average 3.5 times, since January 2015.

4.3 What domain of projects rely the most on `unsafe`?

Approach: We want to understand whether there is a difference in the domain of projects that use `unsafe` sporadically, against projects that rely more heavily on breaking type-safety. Hence, after categorizing the projects' domain, we group the projects that use `unsafe` into two groups: in the first group we include all projects that use `unsafe`, and in the second group we focus on projects that rely heavily on `unsafe`, i.e., the 69 projects containing more than 100 `unsafe` call-sites.

Results: Figure 5 shows the distribution of project domains of all projects in our dataset (left), percentage of projects per domain that use `unsafe` (middle) and the percentage of projects that contain more than 100 `unsafe` call-
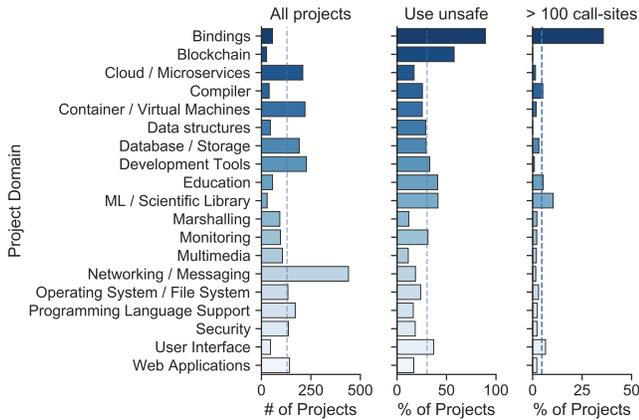
Fig. 5: Distribution of the number of projects per domain (left), accounting for projects that depend on `unsafe` (middle), and projects that rely heavily on `unsafe`. We highlight the averages across all project domains as dashed lines.

sites (right). First, we observe that our dataset contains a predominance of Networking/Messaging projects in our dataset (440 projects), followed by Development Tools (227 projects) and Container/Virtual Machines (220 projects). As our results show, **projects from 20 different domains make use of `unsafe`**. The domains with the highest percentage of projects using `unsafe` are Bindings (89%) and Blockchain (57%) projects, followed by ML/Scientific Library (41%). Several other domains that are more directly related to providing tools for software development, such as Development Tools (33%), Monitoring (31%), Database/Storage (29%), and Data Structures (29%), have more than a quarter of their projects relying on `unsafe` to some extent.

Most notably, **projects that rely heavily on `unsafe` (more than 100 call-sites) represent 35% of all Binding projects in our dataset, the highest share in any category.** Binding projects are projects that aim at bridging the Go language to libraries and platforms written in different programming languages. These projects, such as Gotk3 [40] a bindings for the Graphical Interface framework GTK [30], integrate with platforms not written in Go and use `unsafe` to implement functions that communicate with operating systems and C code. Aside from Bindings projects, other domains have a handful of projects with more than 100 `unsafe` call-sites, such as Networking (7 projects) and Database (6 projects) domains. Example of Networking and Database projects that rely heavily on `unsafe` are the Networking and Security service Cilium [60] and the cloud-native SQL database CockroachDB [19]. Interestingly, only two domains do not have projects with more than 100 `unsafe` call-sites, which are Data structures and Blockchain.

> **Summary of RQ1.** Almost a quarter (24.28%) of most popular Go projects in our dataset use `unsafe` directly. The number of packages using `unsafe` accompanies the projects growth, with an average of 17.5% of packages using `unsafe`. In addition, project from several domains use `unsafe` to some extent, but Binding projects tend to rely heavily on `unsafe`.

TABLE 3: The `unsafe` operations ranked by their usage in 592 Go projects.

Operations	Projects		Call-sites	
	#	%	#	%
type `Pointer`	570	96.28	177,192	97.85
function `Sizeof`	216	36.49	3,527	1.95
function `Offsetof`	21	3.55	312	0.17
function `Alignof`	11	1.86	43	0.02

5 WHAT IS `UNSAFE` USED FOR?

Motivation. Thus far, we saw that almost a quarter of projects in our dataset make use of `unsafe` directly in their code, which bears the follow-up question: what are developers aiming to accomplish by breaking type safety in Go? Knowing the most common usages of `unsafe` is essential for language designers to define better safer alternatives and improve documentation. This knowledge is also important for tool developers to focus their effort in designing solutions (e.g., static analyzers) to the most frequent cases. We analyze this question under two aspects:

1) **Features:** What are the most used `unsafe` features? We want to identify the `unsafe` type and functions most frequently used in practice.
2) **Usage Patterns:** What developers use `unsafe` for? In this aspect, we focus on identifying and understanding the high-level usage patterns most commonly adopted in the most popular Go projects.

5.1 What are the most used `unsafe` features?

Approach. The `unsafe` package is very compact, composed of one type definition (`Pointer`) and three exported functions (`SizeOf()`, `OffsetOf()`, `AlignOf()`). To analyse the features used by developers, we group the analysis of `unsafe` usage conducted in the previous question by the four API components.

Results. As shown in Table 3, the **`unsafe.Pointer` is the most used feature from the `unsafe` API**, used by 96% of projects that rely on `unsafe` and making the bulk of 97% of all `unsafe` operation call-sites in our dataset. The unsafe functions are used less frequently, representing together just 2.24% of all `unsafe` operations call-sites, but are still used by a considerable number of projects. The function `Sizeof`, used to retrieve the size of a variable type, is used by 36% of all projects that rely on `unsafe`.

5.2 What are the most common `unsafe` usages?

Approach: Now, we proceed to understand the usage patterns of `unsafe` employed by Go projects. From the 598 projects that use `unsafe`, we randomly select 270 to perform the manual analysis (see Section 3.4). This number of projects provides us with a representative sample of the 598 projects that rely on unsafe with 5% confidence interval at 95% confidence level. The first two authors labeled one file per project with its dominant use-case.

Our manual analysis yielded the hierarchical label structure shown in Table 4, with six major groups of usage patterns. We aimed to be as specific as possible in our labeling, so if a *Communication* use-case can be attributed to a more specific category, say *System Calls*, we label the

TABLE 4: Usage patterns of `unsafe`, manually identified in 270 project files.

Usage Patterns	Description	Frequency	
Communication	**Communicate with platforms and programs not written in Go**	**45.70%**	
└── System Calls	Using `unsafe` to send parameters to system calls	27.72%	
└── CGO	Using `unsafe` to integrate with C code via `cgo`	17.98%	
Efficient Casting	**Perform more efficient type/array casting**	**23.60%**	
└── Type Casting	Bypassing type-checking and memory copy for performance	19.48%	
└── Marshaling	Using `unsafe` to efficiently (de-)serialize json files	4.12%	
Reflection	**Access object's metadata**	**3.75%**	
Atomic operations	**Writing atomic operations with `atomic/sync`**	**3.37%**	
Address Manipulation	**Using `unsafe` to get memory addresses and copy memory**	**3.37%**	
└── Pointer Arithmetics	Perform pointer address arithmetics	2.25%	
Size of Object	**Getting object size in memory**	**6.74%**	
└── Getting Architecture Info	Inspecting the cpu architecture through `sizeOf(int)` method	4.49%	
Unclear	**Unclear usage-patterns**	**3.00%**	
Others	**Other use-cases (project specific)**	**8.24%**	

file usage as dominated by *System Calls*. In the cases where the specific category could not be identified, we label them as their respective super category. If neither examining the code or the commit messages we encountered sufficient information on the reason of the `unsafe` usage, we labeled the file as Unclear. We evaluate the labeling agreement with the Cohen-kappa interrater reliability [55]. Cohen-kappa inter-rater reliability is a well-known statistical method that evaluates the inter-rater reliability agreement level for categorical scales. The result is a scale that ranges between -1.0 and 1.0, where a negative value means poorer than chance agreement, zero indicates exactly chance agreement, and a positive value indicates better than chance agreement. It is also important to notice that to calculate the inter-rater reliability level, we only consider the exact matches of the most specific labeling (leaf nodes in the hierarchy), but grouped the project-specific cases into a single "Other" category before the agreement calculation. In our analysis, we found that both authors have substantial agreement (kappa=0.65). Finally, the two annotators discussed the divergencies to reach a consensus.

Results: Our findings, depicted in Table 4, show that **developers mostly use `unsafe` for Communication to routines not written in Go (45.70%) and to perform more efficient type casting in their programs (23.60%)**. Other patterns appear less common, such as inspecting the size of objects in memory (6.74%), using `unsafe` to inspect object's metadata through reflection (3.75%), to perform atomic operations (3.37%) and general memory manipulation (3.37%). Also, in 3% of the cases, we could not pinpoint a usage pattern based on the analysis of a single file, and in 8.24% of the cases we deemed the usage pattern too project specific to be discussed here. In the next paragraphs, we dive in details on each group of `unsafe` pattern and explain, with examples, the likely rationale behind the breaking type-safety decisions.

Communication (System Call and CGO.) The most common usage pattern in our sample of 270 projects' files is the usage of `unsafe` as a mean to communicate to routines outside of Go language, which cannot be achieved in a type-safe manner. The communication to systems and programs outside of Go language requires developers to specify the memory address in which such programs can read and

Listing 1: Example of using `unsafe` to set the name of a process

```
// Unsafe pointer as a reference to a target name
ptr := unsafe.Pointer(&name_in_bytes[0])

// Setting a process name with the pointer ptr
_, _, errno := syscall.RawSyscall6(syscall.SYS_PRCTL,
    syscall.PR_SET_NAME, uintptr(ptr), 0, 0, 0, 0)
```

Listing 2: Example of using `unsafe` to call a function in C.

```
func SetIcon(iconBytes []byte) {
    // Convert to a C char type
    cstr := (*C.char)(unsafe.Pointer(&iconBytes[0]))
    // Call the function from systray.h
    C.setIcon(cstr, (C.int)(len(iconBytes)))
}
```

write, to specify parameters and receive their returned objects. For system calls, the `syscall` package [37] offers an API to different operating systems and often requires a `uintptr` with the address of a Go variable as parameters, as illustrated in Listing 1.

In turn, the package `cgo` offers a similar set of API for developers that need to integrate with C code [35]. Similarly to the system call use-case, programs that call C code need `unsafe` to write and read arbitrary memory and communicate with C code, as illustrated in Listing 2. Furthermore, developers also cannot rely on the Garbage Collector from Go to release their C variables, and need to explicitly call the `C.free()` function to release the memory back to the system.

A possible explanation for the high frequency of patterns related to communication could be attributed to projects implementing Bindings. To investigate this, we computed the frequency of patterns with and without the Binding projects and noted that our results still hold: the share of communication related patterns in non-binding projects changed slightly from 57.70% to 45.02%. This shows that the use of `unsafe` as a method for interacting with other systems is prevalent also in non-Binding projects.

Efficient Casting. In 23.60% of the cases, developers use `unsafe` as a method of bypassing compiler checks and memory copy when casting a variable to a different type. Naturally, while there are type-safe ways of converting between valid types, developers only opt for using `unsafe`

Listing 3: Example of unsafe conversion of bytes to string.

```
func String2Bytes(s string) []byte {
    sh := (*reflect.StringHeader)(unsafe.Pointer(&s))
    bh := reflect.SliceHeader{
        Data: sh.Data,
        Len:  sh.Len,
        Cap:  sh.Len,
    }
    return *(*[]byte)(unsafe.Pointer(&bh))
}
```

Listing 4: Code snippet of using `unsafe` as a method to generate hash key, taken from the project Olric.

```
func (db *Olric) getHKey(name, key string) uint64 {
    tmp := name + key
    return db.hasher.Sum64(*(*[]byte)(unsafe.Pointer(&tmp)))
}
```

to optimize runtime and memory allocation of certain functions. The most common case of performance optimization is related to converting string to bytes and vice-versa. Strings are immutable in Go, hence a regular type casting from a string variable requires copying the variable before the cast. To bypass this variable copy, developers use `unsafe` to change the representation of a string into a slice of bytes as illustrated in Listing 3. Since a slice of bytes can be mutated, this operation breaks the immutability of Strings as specified by the Go language, which can have far reaching consequences.

Another particular use-case for `unsafe` under this category is the more efficient marshaling functions, which accounts for 4.12% of the use-cases. The standard marshaling functions are general-purpose and use reflection to identify the object to be marshalled, an operation that can be considered slow in performance critical applications. By using `unsafe`, developers are able to implement their own customized and more efficient marshaling functions.

Reflection. Reflection allows developers to inspect and modify the metadata of types at runtime, simulating some of the dynamism of dynamic typed languages. The `reflect` package [36] requires developers to import `unsafe` to dereference an object accessed through reflection to a pointer. This is done by design, to enforce developers to import `unsafe` when performing such unsafe operations and to prevent `reflect` from replicating some of the functionality of the `unsafe` package.

Atomic Operations Another use-case that occurred in 3.37% of the cases is related to the package `sync/atomic` [34]. The package provides low-level atomic functions for synchronization algorithms. As the documentation of the package poses, using `sync/atomic` properly requires great care from developers to be used correctly. This package offers a way to perform atomic operations with high-performance and without any memory allocation. Currently, there is no safe alternative to perform compare and swap operations on Go objects without memory allocation [51], which explains why the package gives support to `unsafe.Pointer` as opposed to a type-safe alternative (e.g., `interface`).

Address Manipulation. In 3.37% of the use-cases, developers use `unsafe` to get the memory addresses and perform some arithmetic function. Most of the cases we identified are related to using the memory address as a component of a hash function, similarly to the way Java implements the object `hashCode()` mechanism.

Size of Object. In 6.74% of the cases, the `unsafe` package

Listing 5: Code snippet of using `unsafe` to infer the system architecture, taken from the project Telegraf.

```
// Verifying the size of an integer i
if unsafe.Sizeof(i) == 4 {
    is32Bit = true
} else {
    is32Bit = false
}
```

is used to retrieve information about the object size, a use case that is often performed to get the architecture of the underlying system, such as CPU architecture and the system endianness. For instance, in Listing 5 we present a snippet where developers verify the size of an integer `i` to infer whether the operating system is 32-bits or not.

> **Summary of RQ2:** The bulk of `unsafe` usages are related to low-level routines that communicate to operating systems and C code (45.70%), and to improve the performance of type casting (23.60%). Less frequently, developers use `unsafe` to inspect the CPU architecture (4.49%), inspect object's metadata at runtime (3.75%), perform atomic operations (3.37%) and to manipulate memory addresses (3.37%).

6 WHAT ARE THE RISKS OF USING UNSAFE?

Motivation: In the previous RQs, we found that `unsafe` is commonly used in Go projects and there are six main reasons that motivate developers to break type safety. While using `unsafe` is dangerous by definition, we want to investigate the potential risks that using this package entails to software projects in terms of software issues. These potential risks are relevant for Go developers, so that they can identify the types of issues their project are likely to face when breaking type safety. Documenting the risks is also useful for tool developers and researchers, who can focus on strategies of mitigation, identification, and repair of these issues. Therefore, we investigate the risks of using `unsafe` under the following two aspects:

1) **Invalid usage:** We analyze the `unsafe` call-sites with static analyzers to identify invalid pointer conversions in the projects' source code. Invalid pointer conversion is a well-known pitfall that projects depend on `unsafe` are at the risk of falling into.
2) **Unsafe-related issues:** We extract and classify issues related to `unsafe` from the projects repositories. This analysis will give us in-depth insights of real problems faced by projects that use `unsafe`.

6.1 How common is invalid pointer conversion in projects that use `unsafe`?

Approach: We run the tool Go Vet on all projects that use `unsafe`. We were able to automatically analyze 221 projects, due to problems during the project build. Several projects use `unsafe` to integrate with external systems (C code and Operating Systems), hence, these projects in our dataset need external dependencies which cannot be automatically resolved with our building process. Once we run the tool Go Vet, we count the number of reported cases of invalid pointer.

TABLE 5: Number of projects with invalid pointer conversions. We did not observe projects with a mix of false-positive and true-positive invalid pointer conversions.

Statistics	#	%
All projects analyzed	221	100.0%
Projects with invalid pointer conversions	14	6.3%
Projects with reported false-positives	2	0.9%

Results: Regarding invalid usage of `unsafe`, initially the Go Vet tool reported invalid pointer conversions in 16 out of 221 analyzed projects (Table 5). Upon close inspection, we found that cases found in two projects were false-positives, as the conversion to `uintptr` and back to pointer occurred in the same expression, as permitted by the official `unsafe` documentation [39]. Therefore, **14 out of 221 projects (6.6%) had clear invalid pointer conversions**, with functions receiving a `uintptr` as parameter and converting it back to pointer inside the function or performing unsafe pointer arithmetics. Albeit occurring in a minority of the investigated projects, this analysis shows that some popular projects still fall into the most well-documented pitfalls of using `unsafe`. We note that these results should be interpreted as a lower bound of invalid point conversions in these 211 projects. That is, because Go Ver uses a set of patterns to identify invalid pointer conversion, the tool will likely miss cases that do not fall into the known patterns. As a consequence, there could be more than 14 cases in the 221 analyzed projects, but there is no practical way to assess how many of such cases were not retrieved by the tool (false negatives).

6.2 What kind of unsafe-related issues projects that use `unsafe` have?

Approach: After filtering issues through the keyword search in the title ("unsafe", "un-safe"), we identify 286 issues from 119 projects. Our manual inspection revealed that only 103 unsafe-related issues from 63 projects were in fact, related to the `unsafe` package. We conduct our manual analysis on this set. Again, we use Cohen-kappa inter-rater [55] to evaluate the labeling agreement between the two annotators. We found that both annotators had a moderate agreement (Cohen-kappa=0.55) after the first labelling round. Later, both annotators discussed all divergences and reached a consensus in the classification.

Our manual classification of the unsafe-related issues yields a scheme shown in Table 6. We group the issues into 10 categories comprised of Bug Fixing and Project Maintenance issues. We consider as Bug Fixing the issues that were open due to runtime errors or bugs found in the project code base. Project maintenance, on the other hand, are issues created with the goal of improving the project, by adding new features or refactoring the code to improve the overall quality and reduce maintenance costs. In the next paragraphs we discuss, with examples, each category of unsafe-related issues identified in our dataset of the 63 projects.

Unsafe Restriction. The most common unsafe-related issue in our dataset, found in 20 projects, is related to external environmental restriction of the use of the `unsafe` package. In most cases, this is related to the Google App Engine, a platform for cloud development that restricts the usage of `unsafe` for any Go code running in the platform [18], due to safety reasons. For example, a developer in the project GJSON wrote: *"I wanted to use this package within a Google App Engine project, and due to package "unsafe" being used, it is not compatible"* [43]. In many of these cases, the solution found for project maintainers was to provide a version of their package without `unsafe` dependency, or to remove the `unsafe` dependency in favor of a safer alternative. However, we also found cases where the usage of `unsafe` is so widespread in the project's code, the maintainers were not willing to remove it. In such cases, there is an encouragement that users fork the project repository to create a new safe-version of their packages, e.g., this developer wrote: *"The atomic swap is used all over the place during transactions, so I don't think we'd want to take a change that removes it, but you could make a fork and replace unsafe with a mutex basically to make it less performant but safe"* [46].

Runtime Errors. In 16 projects, we encounter issues that were created due to runtime errors caused by the misuse of `unsafe` package. The most common type of runtime error, found in 6 projects, is related to crashing errors due to bad pointers, e.g. *"Prometheus crashes and hangs on 'fatal error: found bad pointer in Go heap"* [17]. Such errors can be caused by the Garbage Collector releasing unintended variables, mismanagement of operations that read and write memory and possibly other causes. Another cause of crashing bugs in some projects were related to the conversion between different type layouts. As a developer points out in an issue *"I suspect the problem is that there is no guarantee that the alignment is correct after the pointer conversion"* [32]. The misuse of `unsafe` has also been reported to cause data corruption, *"Unsafe use of unsafe that leads to data corruption"* [61] due to breaking string immutability in the string to bytes conversion. Furthermore, we also found reports indicating that the program did not crash during execution but the wrong usage of `unsafe` has led the program to produce wrong results: *"The combination of this version of siphash and use of unsafe.Pointer to obtain a byte slices caused back-to-back ast#Term.Hash calls to return different values!"* [62]. These errors are difficult to diagnose and replicate: 5 of such issues mentioned problems to reproduce the runtime error due to the non-deterministic nature of the problem. As a developer describes: *"But someday, when using my JS package, I stumbled upon an unexpected behavior in one Lua function[4]it is not even an error, just that the string.gsub function isn't behaving correctly"* [28]. All such reports corroborate with the expected risks of breaking type safety, which may cause crashing errors, data corruption, wrong behavior and in many cases are difficult to diagnose and replicate.

Wrong Usage. We group in this category, 14 bugs found on 12 projects caught by maintainers or collaborators while inspecting the code, with no report of runtime issues. The most common case, found in 5 projects, is related to slice conversion: *"While not likely to occur in the wild, changes to how GC inlines functions in 1.12 creates the possibility for data loss when serialization_littleendian.go uses unsafe to change the*

---

4. The Lua function was implemented in Go, through gopher-lua.

TABLE 6: Issues related to `unsafe` identified in 63 projects. We present both the number of issues (#) and the number of projects in which they occur (# proj.).

Task	Issue category	Reason behind the Issue	#	# Projects
Bug Fixing	Unsafe restriction	Deployment environment restricts the use of `unsafe`	30	20
	Runtime errors	Wrong usage of `unsafe` caused runtime and crashing errors	18	16
	Wrong usage	Bug found in the code related to wrong usafe of `unsafe` (preemptive)	14	12
	Static/Dynamic Check Violations	Bug in the code found by static code analyzers (Go Vet or Go1.14)	7	6
	Breaking Changes	Issue due to breaking changes introduced in `unsafe`	4	3
	Portability Issues	Program did not work in different architectures	3	2
Maintenance	Remove unsafe	Replace `unsafe` with a safer implementation variant	13	12
	Extending `unsafe` support	Add support to unsafe.Pointer	6	6
	Using `unsafe` for optimization	Optimize code using `unsafe`	6	4
	Isolate unsafe usage	Move `unsafe` code to a dedicated package	2	2

type of slices" [49]. Slice conversions also suffer from invalid pointer conversion issues, having the Garbage Collector release the slice in the midst of the conversion. A commonly employed solution for this problem is to explicitly inform the runtime system to not release the slice header during the conversion, by calling the function `runtime.KeepAlive`. However, `runtime.KeepAlive` is not always a valid solution [7] and can still yield memory corruption in some particular cases [22]. We also found issues related to traditional invalid pointer conversion, as described by a developer in an issue: *"The following code is invalid because it puts a non-pointer value into a pointer-type, if the GC finds this non-pointer it will crash the program"* [11]. Furthermore, a particular issue related to `unsafe` usage draws our attention. The issue was reported as a security related in the project `nuclio`, where convertion from string to bytes raised the possibility of exposing parameters from an internal library, as a developer reports: *"Without going through their code this [issue] is likely due to buffer reuse, which is a race condition at the least and a security issue at the worst"* [66].

Static/Dynamic Check Violation. In total, 7 issues from 6 projects were opened due to static and dynamic checks identifying suspicious usage of `unsafe`. In 5 out of 7 cases, the issue was raised by the newest Go 1.14 version (released in February, 2020) which incorporated in the compiler a more robust invalid pointer conversion checker, which instruments the code to find violations to the `unsafe` rules dynamically. For example, a developer reported: *"Go 1.14 introduces new runtime pointer checking, enabled by default in race mode which verify that the rules for using unsafe are followed"* [44]. This check is performed if the program is compiled with a race detector (flag `-race`) in Go 1.14 and emits a fatal error if a violation is found, forcing developers to fix the problem. The other two cases were motivated by `go vet`, the static analyzer we used to identify invalid pointer conversions in our previous analysis.

Breaking Changes and Portability Issues. While less frequent, we also found issues related to breaking changes (3 projects), and related to portability issues (2 project). For instance, in 2011 a change in the function `sizeOf` of the package `unsafe` created several bugs in a project. As a developer describes *"In tip, unsafe.Sizeof has been changed to return uintptr. This causes a variety of issues during the build of walk"* [64]. We also found portability issues related to projects that use `unsafe` when used in different Operating Systems. For instance, in one issue a developer reported that *"unsafe (marshaler and unmarshaler) test fails on big endian*

architectures" [65], indicating unsafe implementation worked on powerpc (32-bit) but did not work on s390x (64-bit) architectures. These issues corroborate with the `unsafe` package documentation, which clearly states that portability and compatibility guarantees do not hold when using `unsafe`.

Project Maintenance. We report Project Maintenance issues related to `unsafe` grouped in four groups. Such issues are not really bugs found in projects and do not represent a direct risk for the projects. Instead, they represent tasks that are open by collaborators to improve project maintenabililty and are reported here in our study for completeness. The most common issue, found in 12 projects, were created to suggest the removal of `unsafe` to mitigate the risks of using `unsafe`, e.g., *"Our benchmarks also show that there is no significant difference between safe and unsafe. This allows to remove optimizations with unsafe and simply rely on plain code generation"*. [63]. Moreover, we found 2 projects with issues related to isolating the `unsafe` dependency to a reduced number of packages in a project. This indicates that some developers are keen to reduce the reliance on `unsafe` due to its inherent risks. While there is an effort to keep `unsafe` usage at minimal levels in some projects, we also found that developers expand their API to support the `unsafe.Pointer` type ("Support for channels, maps and unsafe.Pointers" [8]). As shown in RQ2, developers frequently rely on `unsafe` for optimizations and we found issues open in 4 projects proposing a refactoring to more efficient code that uses `unsafe`.

Fix-patterns to unsafe related issues. We investigated the recurrence of fix-patterns that could be used to fix unsafe related issues automatically. We focused in identifying fix patterns related to issues of Wrong Usage and Runtime Errors, as these are closer to the traditional functional bugs. From the 32 issues, we found the majority of 24 issues to be entirely context-dependent. These fixes were related directly to the business logic of the program and are hard to generalize. The remaining 8 issues were fixed following fix-patterns related to unsafe type-casting, which we discussed in Section 2: developers should perform operations using a single expression to avoid having the GC release variables in use.

All the examined bug issues are exclusively caused by the use and misuse of `unsafe`. The main takeaway from this analysis is that projects that do not break type safety and depend on type-safe third-party packages are free of encountering the issues we discussed in this analysis, such as crashing errors caused by bad pointers, have portability

and compatibility issues or have their code restricted to deploy in different environments.

> **Summary of RQ3:** We found suspicious cases of invalid pointer conversions in 14 out of 211 projects. More importantly, projects that use `unsafe` face several unsafe-related issues, from having their deployment restricted (20 projects), experiencing runtime errors that are hard to reproduce (16 projects), introducing bugs due to misuse of `unsafe` (12 projects).

7 DISCUSSION

In this section, we delve into the design decisions behind the exposition of `unsafe` on standard Go packages (Section 7.1) and discuss the performance benefits of common unsafe optimizations (Section 7.2). Then, based on our results we discuss a set of recommendations to improve the safety of the language (Section 7.3) and describe an exchange we had with the Go team about our study (Section 7.4).

7.1 Design decisions of APIs exposing `unsafe`

The majority (56%) of the `unsafe` usage patterns are linked to packages from standard libraries (`syscall`, `cgo`, `reflect` and `atomic`). Given the risks associated with breaking type safety, we wanted to investigate why standard Go packages exposed `unsafe` in their APIs. Hence, we went through the list of standard packages in Go[5], and listed the packages that expose `unsafe` by either using `unsafe.Pointer` or `uintptr` in their API. The standard library of Go has a total of 147 packages, covering a variety of different functionalities, from the language core to marshaling of different data formats. We found that **7 out of 147 packages expose `unsafe` in their APIs (see Table 7)**. We observe that these seven packages expose `unsafe` for the following reasons:

R1 The package provides access to low-level implementation which needs to be handled carefully by developers. In this case, having the requirement of importing `unsafe` forces developers to understand the unsafeness of the package itself. This is the case of the `reflect` package, which could return an unsafe pointer in the function `Pointer()`, but instead returns an `uintptr` to force developers to import `unsafe` to do the conversion [36].

R2 The package provides low-level pointer operations that cannot be efficiently implemented with safe pointers. This is the case of the package `atomic` that performs atomic operations in primitives and pointers. It is not possible to dereference or alter the address of a safe pointer in Go, hence the package needs to expose `unsafe` to swap pointers in a fast and atomic way [51].

R3 The package gives access to non-portable or system-specific implementation, such as `syscall` [37] and `cgo` [1]. In both cases, the reliance on syscalls and C code already puts the program into non-compatibility territory, as operating systems and external dependencies might change their APIs without further notice.

Overall, we find that only a small share of standard Go packages requires the use (or import) of the package `unsafe`.

5. List of standard packages in Go: https://golang.org/pkg/

TABLE 7: Packages from Go standard library that have at least one function that requires the import of `unsafe`. We describe the reasons R1, R2, and R3 in the text of Section 7.1.

Package	Description	Reason
os	Provides a platform-independent interface to operating system functionality.	R1, R3
reflect	Implements run-time reflection, allowing a program to manipulate objects with arbitrary types.	R1
runtime	Contains operations that interact with Go's runtime system.	R1
cgo	Runtime support for code generated by the cgo tool.	R3
debug	Contains facilities for debugging programs.	R1
atomic	Provides low-level primitives for implementing synchronization algorithms.	R2
syscall	Contains an interface to the low-level operating system primitives.	R3

TABLE 8: Example of pull-requests open to improve the performance of functions by using `unsafe`.

Optimization	Reported performance improvement	Ref.
Slice casting	60% to 90% faster execution time	[33]
String to Bytes	0% to 40% less memory allocation	[57]
String to Bytes	50% less memory allocation	[58]
String to Bytes	25% faster exec time + 27% less allocations	[56]

The exposition of `unsafe` from such packages seems to be driven by a design decision to warn developers of the risk of certain operations as well as to provide the flexibility and performance that cannot be achieved in a type-safe manner with the current Go language.

7.2 Performance benefits of `unsafe` casts

Developers use `unsafe` as a method for implementing faster functions in 23.6% of the usages clasified in RQ2. We want to investigate level of performance gain that can be obtained by breaking type safety in Go. We looked into the six issues classified as "Using `unsafe` for optimization" in Section 6.2 and identified that in four cases, the developers reported the optimization obtained by using `unsafe`. We report these in Table 8. The improvement observed varies from no improvement in some functions to up to 90% execution time and 50% less memory allocation. While these number seem impressive, they report the time spent and memory allocated on the function optimized alone (micro-benchmarks). We found no end-to-end performance tests to evaluate how much of an impact an improvement would have on a software in production.

As we discussed in Section 5, the most common code unsafe function used to optimize code is the conversion of strings to bytes and vice-versa. This is a well-known method for optimizing performance but has harmful potential consequences of breaking string immutability specified by the Go language. For instance, trying to modify the bytes after the conversion from a string will cause a segmentation violation error. In fact, most code snippets come with a recurring comment of "Use at your own risk" indicating that developers are aware of the risks but choose to use this as an optimization method. To illustrate the performance benefits of this optimization method, we crafted two micro-benchmarks to compare the performance of 1) safe vs unsafe

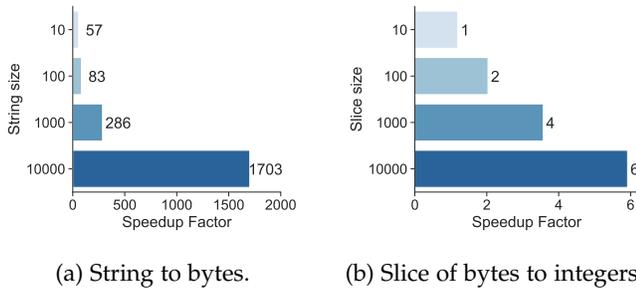(a) String to bytes. (b) Slice of bytes to integers.

Fig. 6: Performance speedup factor obtained by using `unsafe` casting compared to type-safe alternatives in a 2.3 GHz Quad-Core Intel i5 machine. In both cases, the time and memory allocation complexity of the type-unsafe alternative is O(1) compared to the O(n) of the safe implementation.

string conversion and 2) safe vs unsafe slice conversion. We run the benchmark using Go native harness benchmarking package [38] to reliaby measure the steady-state performance of these functions under different loads. We show in Figure 6a the speedup obtained in a micro-benchmark with varying string sizes when converting string to bytes. Even at smaller string sizes (10 characters), the speedup obtained through bypassing the string copy reaches a factor of 57x, which may explain why so many popular Go projects contain this function in their code-base.

Developers also use `unsafe` to speed-up slice casting operations. As discussed in the Wrong Usage issue in Section 6.2, this method also has inherent complexities. If not done correctly the conversion can cause the program to crash, particularly if the types converted do not have the same memory layout. We also craft micro-benchmarks to evaluate the performance speedup of using the `unsafe` cast and found that, unlike string to bytes conversions, the performance benefits are not clear-cut (see Figure 6b). Unsafe slice conversions are up to 6x faster on a slice with 10 thousand integer elements, which may not justify the risks of the type-breaking conversion in this particular case.

7.3 Recommendations

Our results show that `unsafe` usage is widespread, motivated by low-level software integration and performance optimization. However, it puts projects at the risk of several issues. Next, we draw a series of recommendations from our results that could help make Go language more safe.

Inclusion of more powerful static analyzers. Currently, the tool Go Vet focus on identifying invalid pointer conversions, one of the main issues with using `unsafe`. However, researchers and practitioners can develop analyzers that attempt to find other invalid cases of `unsafe` usage, such as using `unsafe.Pointer` to dereference a nil pointer or access a memory address beyond the allocated memory space. For instance, statically identifying null pointers is a problem well investigated in languages like C and C++ [25], [42], and similar approaches could be employed in Go to identify unsafe nil pointer conversion. In fact, the newest Go release (1.14) already provides a more robust set of checks, by embedding a compile option that instruments the code to capture violation of safety rules.

Furthermore, our study shows that the majority of `unsafe` usages are concentrated on a handful of patterns.

The catalog of usages provided by our study can be employed in static analysis tools to identify misuses of widely-used patterns. For instance, two widely-used patterns, the slice and bytes-string conversions should draw special attention and become target patterns for static analyzers. Developers often implement such conversions without guarding for invalid pointer conversion, causing their programs to crash and behave erroneously.

Improve documentation on frequent `unsafe` usages. Our investigation showed that the conversion of string to bytes (and vice-versa) and slice types are a very common optimization method (present in at least 19% of projects as shown in RQ2). However, the official documentation of `unsafe` only briefly touches the fundamentals of the issues developers encounter when writing this code. This is further corroborated by discussions in mailing lists related to Go on how to properly convert string to bytes [31] and proposals to clarify the usage of `unsafe` when performing syscalls [24]. While Go language maintainers have expressed that including such conversions in the standard API is not desired [50], due to the issues of breaking string immutability, an official statement on how to properly convert between string and bytes could help developers at writing correct code conversion and finding the ill-implemented variants.

Inclusion of Generics. We found some usages of `unsafe` to be primarily motivated by the type flexibility that a support for generics would provide, e.g., Marshaling which is present in at least 4.12% of the projects. Without generics, generic functions (e.g., customized ways of sorting a slice) are accomplished with interfaces, reflection or code generation [12]. The standard marshaling package relies on runtime reflection to encode and decode json to Go structs, which can negatively impact the performance of object serialization. The inclusion of generics would allow for more efficient encoders/decoders without relying on `unsafe`, as the compiler could generate code automatically based on type specified by developers in their generic functions [9].

Planning breaking changes to `unsafe` in the Go language. There exists several proposals that if implemented in Go 2.0 may impact programs that currently use `unsafe` [41], [67], introducing breaking changes in the language. Our study shows that a quarter of the most popular Go projects use `unsafe` in their code, and given their notoriety, it is expected that such projects are used by a large share of the Go community. We build a dependency graph using the Go List command, considering only the project in our dataset (2,438 most popular Go projects), and found that 40% of the projects either use `unsafe` directly or depend directly on a project that uses `unsafe`.

Fortunately, our study also shows that the majority of the `unsafe` usage in popular Go projects is well-localized in code: most projects concentrate their `unsafe` in at most 2 files. This indicates that the cost for updating the `unsafe` usages to comply to possible breaking changes in Go 2.0 version should be manageable for most projects. Still, language designers could mitigate the cost for adopting the new language version by communicating the adopted proposals in advance to the community, or even better, by employing a tool for migrating some of the `unsafe` usages to the newest package version within the Go Fix command [1]. The Go Fix is a tool created with the sole purpose of migrating old APIs

to new ones in the case of breaking changes in the language, and can be used to update `unsafe` usages from valid Go 1.x code to Go 2.x. Our methodology can also be employed on a larger set of Go projects to assess the impact of the breaking change in the entire Go ecosystem and better aid language designers at planning mitigation measures.

7.4 Perspectives of the Go Team

The end-goal of our work lies in making the Go language safer and easier to use by developers. Naturally, this goal is way beyond what this study can achieve on its own and requires that both the language designers and developers are aware of the risks of using unsafe and adopt measures to improve its usage in the long run. Coincidentally, once we made a pre-print of this work available online, we were reached by a member of the Go Team, Matthew Dempsky, with positive remarks on our study. Matthew is a maintainer of the Go compiler and responsible for the implementation of `checkptr`, the dynamic checker deployed with Go 1.14 reported in Section 6.2. With his help, we circulated the pre-print version of this paper to his colleagues to receive feedback on any result from the paper the Go team deemed to be surprising or useful. We summarize the highlights of this exchange in the following:

1) The widespread usage of unsafe was received as a surprise by the Go team. The usage of `unsafe` by 24% of the projects is significantly higher than the Go team expected. As Matthew stated, *"Other team members were more optimistic that developers would avoid or could implement their project without using package unsafe. I think this result will help dispel that misconception and justify spending more time on making package unsafe easier to use."*

2) The usage patterns and some of the mistakes were known by the Go team and the results were consistent with what the members expected. Still, their relative frequencies provided new insights about their importance. *"I think knowing the frequencies will be useful in prioritizing improvements to unsafe.Pointer."*

3) The Go team has also pointed out that the trends observed in the historical analysis were novel and insightful, showing that projects tend to keep up their `unsafe` dependencies during their entire development.

Overall, the interest and the feedback we received from the Go Team highlight a positive outcome of our study. In particular, the predominance of `unsafe` usage in popular Go projects (RQ1) showed that breaking type safety is common and developers need better support to mitigate its risks.

8 RELATED WORK

There exists a plethora of work that investigate how developers use certain features of the programming language. In this section, we discuss the work that is most related to our study. We grouped prior work into work related to unsafe features in programming language and studies on mining repositories to assess language features.

8.1 Unsafe Language Features

Some prior studies investigates how developers use low-level code features in different programming languages. Nagappan et al. [59] investigate how developers use the `goto`

statements in a representative sample of C programming files. Motivated by the harmful stigma of `goto` statements, the authors qualitatively investigate how such command is used in modern C code. The investigation showed that developers nowadays limit themselves to use `goto` only in very specific circumstances, such as error handling and cleaning up resources, and that it does not appear to be harmful in practice.

Some other studies focus on investigating the usage of dynamic features on programming languages. Mastrangelo et al. [52] investigated how and when developers use casting on thousands of Java projects. The results show that casting is widely-used by Java developers and that half the casts are not guarded locally to ensure against runtime errors. Similarly to our study, this investigation also cataloged a common set of use-case patterns that can help language maintainers and tool developers to better accommodate the most common usages of dynamic casting. Callau et al. [16] investigated how dynamic and reflective features are employed in SmallTalk, by surveying a thousand SmallTalk projects. While SmallTalk is a dynamically-typed language, the authors reported that dynamic features are not widely used by SmallTalk developers. In fact, the two most pervasively used features are the ones other static languages implement, indicating a more conservative approach with seemingly unsafe features than the ones we observed in popular Go projects.

One of the works most closely related to ours is the investigation of unsafe APIs in Java by Mastrangelo et al. [53]. In their work, the authors investigated how developers use the `sun.misc.Unsafe`, a class that exposes low-level and unsafe features, violating Java safety guarantees. Differently from the `unsafe` package in Go, the `sun.misc.Unsafe` API gives access to more than type safety violations, for instance, developers can violate method contracts by throwing a checked exception undeclared by a method. Their investigation found that only a small share (1%) of software artifacts use the unsafe API in their code directly. In contrast, we found that 24% of the most popular Go projects rely on unsafe to some extent, showing that using unsafe APIs seems to be more common in Go than in Java.

Another closely related work investigated the use of Unsafe Rust [27] in a large set of Rust libraries. Unsafe Rust is in fact a language embedded in Rust, but it gives access to similar functionalities as the `unsafe` package, such as C-style pointer operations and access to arbitrary memory locations, which are not guaranteed by the compiler to be safe. The authors have identified that 30% of the Rust libraries use the keyword unsafe and more than half of them depend on another library that is unsafe. This results are in line with the findings we observed in our study. Our study contributes to this body of work, that investigate the practices of breaking type safety [26], [53] in programming languages and provides insights and recommendations to make programming languages more safe.

8.2 Mining Repositories of Usage of Language Features

Several studies empirically investigate how developers use different language features through mining software repositories. Mazinanian et al. [54] mined software repositories to

investigate how Java developers use Lambda in their programs. Similarly, Costa et al. [20] profiled how developers select and tune their data structures in Java programs, showing that developers only rarely tune their data structures and prefer the standard implementations, despite having better performance variants available. Krikava et al. [48] investigate the usage of implicits - a language feature that allows developers to reduce boilerplate code - in Scala programs. The results showed a pervasive use of the feature in Scala projects hosted in GitHub. Guilherme and Weiyi [21] performed an empirical study to examine the prevalence of exception-handling anti-patterns in Java and C# projects. Their findings showed that all studied projects contain exception handling anti-patterns in their source code. Wang et al. [68] studied the evolution regular expressions over time from the different aspects. Their results showed that the use of regular expressions is stable in the development history of the studied GitHub projects.

The aforementioned work mines software repositories to quantitatively assess the level of adoption of language features from developers. Our study shares part of this mining repository methodology, such as selecting studied projects, parsing the code to get quantitative results and manually analyze the usage patterns for a qualitative assessment.

9 THREATS TO VALIDITY

This section describes the threats to the validities of our study.

9.1 Internal validity

Threats to internal validity are related to experimenter bias and errors. To identify `unsafe` usages, we build a customized parser which could miss or introduce false positives in our analysis. To mitigate this threat, we first took special care to identify all possible ways `unsafe` can be used in Go code, by inspecting import statements for `unsafe` in a number of projects. Second, during our manual examination of the detected cases of `unsafe` to catalog the usage patterns, we only found a single case (1/270) of false positive, which give us a the confidence in the accuracy of our customized parser. Developers had defined a custom object named "unsafe" and called a method from the object, filtered in by our parser. Third, during our experiments, we did not observe any errors when conducting both the analysis on the last snapshot nor the historical analysis, which could indicate we could miss real `unsafe` usages (false negatives). Also, our parser is compatible with all Go version since Go 1.0.0. Therefore, we believe the results yielded by our unsafe parser will hold. We also use the tool Go Vet to identify suspicious cases of invalid pointer conversions. While we have manually assessed that 2 out of 16 cases were false positives, the tool could miss real cases due to the complexity of statically analyzing the code. Hence, our results with Go Vet should be interpreted as a lower bound of invalid pointer conversions.

Furthermore, we conduct two major manual analyses in our study, to investigate the most used usage patterns and to classify unsafe-related issues. In the classification of the `unsafe` usage patterns, we include a single random file per project, hence, our analysis is based on the usage of a single randomly selected file from a representative sample of projects. This ensured we performed a cross-project analysis and did not bias our usage pattern catalog towards a minority of projects that use `unsafe` extensively (100+ files). To investigate the representativeness of our analysis, we fully classified all `unsafe` usages of 30 randomly selected projects of our dataset, accounting for 168 files that depend on `unsafe`. This sample analysis showed that 66% of the projects contain only one `unsafe` usage pattern, and 86% of the projects contain fewer than two usage patterns in their code-base. Furthermore, we observed a similar ranking as the one shown in Section 5.2 when analyzing the distribution of usages under a different perspective: the ranking of `unsafe` usages based on the percentage of projects with at least one case of the usage pattern. Upon the results of our validation, we believe the ranking reported in RQ2 holds even under other (reasonable) sampling strategies.

Regarding the analysis of unsafe-related issues, it is important to note that the categories of issues obtained in this analysis are not exhaustive or necessarily representative. We only investigate issues that have the keyword "unsafe" in its title and hence are bound to miss many unsafe-related issues. Hence, our analysis is a lower bound of possible unsafe-related issues, providing qualitative support to the risks of using `unsafe` entails to software projects.

9.2 External validity

Threats to external validity are related to the generalizability of our findings. Our investigation focused on the most popular Go projects. Go has established itself as a programming language for high-performance infrastructure projects and the projects in our dataset reflect that (e.g., there is a relative low number of front-end projects). The `unsafe` package is used for low-level implementation and optimization, hence its prevalence is expected to decrease on less popular projects or on a more diverse set that may not have the pressure for high-performance code. We argue that, while not fully generalizable to all Go software projects, our dataset contains the most influential projects of the current Go landscape. Also, in our analysis, we examine open-source Go projects that are hosted on GitHub. Thus, our results may not generalize to proprietary projects.

10 CONCLUSION

In this paper, we present the first study on the usage of `unsafe` in Go programs. We conduct a mix-method analysis of the prevalence of `unsafe` in popular open-source Go projects, investigate what developers aim to achieve by breaking type safety, and evaluate some of the real risks projects that use `unsafe` are subjected to. Our results have shown that the use of `unsafe` is prevalent, with one in four Go projects breaking type safety primarily for communicating with programs outside of Go language (C code and Operating Systems), and optimizing type casting functions. Developers have made the conscious effort of keeping `unsafe` restricted to a selected set of packages in their project, but report a series of unsafe-related issues due to misusage of `unsafe`. Their projects may have the deployment restricted by shared cloud environments such as the Google App Engine, developers report crashing errors

and wrong results due to invalid pointer conversion, and in some cases the program produces the wrong behavior due to non-deterministic causes.

Our study can be used as empirical evidence on the state of usage of `unsafe` in Go and help motivate further tools and language developments that could make Go safer. We suggest that special attention should be given towards creating tools that identify further bad practices related with the package, similarly to how Go 1.14 introduces more robust checks for invalid pointer conversions, as developers seem to struggle in writing valid unsafe code. Language maintainers can also mitigate the encountered issues by documenting official unsafe snippets to be used by the community, in particular on type casting of string and bytes, widely used as a means of optimization. Furthermore, the level of adoption of `unsafe` should also be taken into consideration when planning future versions of the language, as the package as risky as it is, seems to be integral to the Go community.

ACKNOWLEDGMENT

REFERENCES

[1] fix - the go programming language. https://golang.org/cmd/fix/. [Online; accessed on 01/03/2020].

[2] Go 1 and the future of go programs - the go programming language. https://golang.org/doc/go1compat. [Online; accessed on 10/11/2019].

[3] The go project - the go programming language. https://golang.org/project/. [Online; accessed on 03/02/2020].

[4] vet - the go programming language. https://golang.org/cmd/vet/. [Online; accessed on 10/11/2019].

[5] What are the possible consequences of using unsafe conversion from []byte to string in go? https://stackoverflow.com/questions/33952378/. [Online; accessed on 18/12/2019].

[6] Stack overflow developer survey 2019. https://insights.stackoverflow.com/survey/2019, 2019. [Online; accessed on 10/11/2019].

[7] G. 101. The unsafe article states a runtime.keepalive call is needed in the string2byteslice example. this is wrong. https://github.com/go101/go101/issues/151. (Accessed on 10/14/2020).

[8] A. Arzilli. Support for channels, maps and unsafe.pointers by aarzilli · pull request #286 · go-delve/delve. https://github.com/go-delve/delve/pull/286, Nov 2015. [Online; accessed on 01/03/2020].

[9] T. G. Authors. Go2genericsfeedback. https://github.com/golang/go/wiki/Go2GenericsFeedback. [Online; accessed on 03/04/2020].

[10] T. G. Authors. tools/unsafeptr.go · golang/tools. https://github.com/golang/tools/blob/master/go/analysis/passes/unsafeptr/unsafeptr.go, Oct 2019. [Online; accessed on 03/04/2020].

[11] W. Bitter. Invalid use of unsafe.pointer. https://github.com/go-gl/gl/issues/18, May 2015. [Online; accessed on 03/04/2020].

[12] V. Blanchon. Go: Is the encoding/json package really slow? - a journey with go - medium. https://medium.com/a-journey-with-go/go-is-the-encoding-json-package-really-slow-62b64d54b148, May 2019. [Online; accessed on 01/03/2020].

[13] V. Blanchon. Go: What is the unsafe package? - a journey with go. https://medium.com/a-journey-with-go/go-what-is-the-unsafe-package-d2443da36350, Jun 2019. [Online; accessed on 30/05/2020].

[14] H. Borges, A. Hora, and M. T. Valente. Predicting the popularity of github repositories. In *Proceedings of the The 12th International Conference on Predictive Models and Data Analytics in Software Engineering*, PROMISE 2016, New York, NY, USA, 2016. Association for Computing Machinery.

[15] H. Borges, A. Hora, and M. T. Valente. Understanding the factors that impact the popularity of github repositories. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2016.

[16] O. Callaú, R. Robbes, E. Tanter, and D. Röthlisberger. How developers use the dynamic features of programming languages: The case of smalltalk. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, MSR '11, page 23–32, New York, NY, USA, 2011. Association for Computing Machinery.

[17] I. Chekrygin. Prometheus crashes and hangs on fatal error: found bad pointer in go heap (incorrect use of unsafe or cgo?). https://github.com/prometheus/prometheus/issues/2263, Dec 2016. [Online; accessed on 01/03/2020].

[18] G. Cloud. App engine. https://cloud.google.com/appengine. [Online; accessed on 01/03/2020].

[19] CockreachDB. Cockroachdb - the open source, cloud-native sql database. https://github.com/cockroachdb/cockroach. [Online; accessed on 01/03/2020].

[20] D. Costa, A. Andrzejak, J. Seboek, and D. Lo. Empirical study of usage and performance of java collections. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, ICPE '17, page 389–400, New York, NY, USA, 2017. Association for Computing Machinery.

[21] G. B. de Pádua and W. Shang. Studying the prevalence of exception handling anti-patterns. In *Proceedings of the 25th International Conference on Program Comprehension*, ICPC '17, page 328–331. IEEE Press, 2017.

[22] M. Dempsky. The go playground - keepalive not safe. https://play.golang.org/p/W8XYZjW0t-e. (Accessed on 11/02/2020).

[23] M. Dempsky. Guarantees for package unsafe. https://bit.ly/39WPORD, Aug 2016. [Online; accessed on 01/03/2020].

[24] M. Dempsky. proposal: unsafe: clarify unsafe.pointer rules for package syscall. https://github.com/golang/go/issues/34684, Oct 2019. [Online; accessed on 01/03/2020].

[25] F. C. Eigler. Mudflap: Pointer use checking for c/c+. In *GCC Developers Summit*, page 57. Citeseer, 2003.

[26] A. N. Evans, B. Campbell, and M. L. Soffa. Is Rust used safely by software developers? In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, page 246–257, New York, NY, USA, 2020. Association for Computing Machinery.

[27] A. N. Evans, B. Campbell, and M. L. Soffa. Is Rust used safely by software developers? In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, page 246–257, New York, NY, USA, 2020. Association for Computing Machinery.

[28] fiatjaf. Strange error happening in just one function of a go library after compiled. https://github.com/gopherjs/gopherjs/issues/642, May 2017. [Online; accessed on 03/04/2020].

[29] S. Fincher and J. Tenenberg. Making sense of card sorting data. *Expert Systems*, 22(3):89–93, 2005.

[30] T. G. Foundation. The gtk project - a free and open-source cross-platform widget toolkit. https://www.gtk.org/. [Online; accessed on 03/02/2020].

[31] Francis. Clarification on unsafe conversion between string and []byte. https://bit.ly/2VHHoJN, Sep 2019. [Online; accessed on 01/03/2020].

[32] S. Frei. Flaky "unsafe pointer conversion"-panics with -race and go1.14rc1. https://github.com/spaolacci/murmur3/issues/29, Feb 2020. [Online; accessed on 01/03/2020].

[33] W. Glynn. Use unsafe/reflect in serialization. https://github.com/RoaringBitmap/roaring/pull/41, Oct 2015. (Accessed on 10/30/2020).

[34] Golang. atomic - the go programming language. https://golang.org/pkg/sync/atomic/. [Online; accessed on 18/12/2019].

[35] Golang. cgo - the go programming language. https://golang.org/cmd/cgo/. [Online; accessed on 18/12/2019].

[36] Golang. reflect - the go programming language. https://golang.org/pkg/reflect/. [Online; accessed on 18/12/2019].

[37] Golang. syscall - the go programming language. https://golang.org/pkg/syscall/. [Online; accessed on 18/12/2019].

[38] Golang. testing - the go programming language. https://golang.org/pkg/testing/. (Accessed on 11/13/2020).

[39] Golang. unsafe - the go programming language. https://golang.org/pkg/unsafe/. [Online; accessed on 18/12/2019].

[40] gotk3. gotk3/gotk3: Go bindings for gtk3. https://github.com/gotk3/gotk3. [Online; accessed on 03/02/2020].

[41] R. Griesemer. proposal: spec: disallow t¡-¿uintptr conversion for type t unsafe.pointer. https://github.com/golang/go/issues/20171. [Online; accessed on 01/08/2019].

[42] D. Hovemeyer, J. Spacco, and W. Pugh. Evaluating and tuning a static analysis to find null pointer bugs. *SIGSOFT Softw. Eng. Notes*, 31(1):13–19, Sept. 2005.

[43] J. K. III. Removed usage of package "unsafe" to allow google app engine compatibility. https://github.com/tidwall/gjson/pull/69, Apr 2018. [Online; accessed on 01/03/2020].

[44] J. K. III. runtime error: unsafe pointer arithmetic (go 1.14 checkptr). https://github.com/yuin/gopher-lua/issues/254, 2018 2018. [Online; accessed on 28/05/2020].

[45] G. Inc. The state of the octoverse — the state of the octoverse celebrates a year of building across teams, time zones, and millions of merged pull requests. https://octoverse.github.com/. [Online; accessed on 01/08/2019].

[46] A. Issa. Google app engine failed parsing input: parser: bad import "unsafe". https://github.com/hashicorp/go-memdb/issues/44, Dec 2017. [Online; accessed on 01/03/2020].

[47] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian. An in-depth study of the promises and perils of mining github. *Empirical Software Engineering*, 21(5):2035–2071, Oct 2016.

[48] F. Křikava, H. Miller, and J. Vitek. Scala implicits are everywhere: A large-scale study of the use of scala implicits in the wild. *Proc. ACM Program. Lang.*, 3(OOPSLA):163:1–163:28, Oct. 2019.

[49] D. Lemire. Keep header.data alive during unsafe operations. https://github.com/RoaringBitmap/roaring/pull/226, Aug 2019. [Online; accessed on 03/04/2020].

[50] T. Liron. Feature: provide no-copy conversion from []byte to string · issue. https://github.com/golang/go/issues/25484, May 2018. [Online; accessed on 01/03/2020].

[51] C. Mastrangelo. proposal: sync/atomic: add swap, compare-andswap. https://github.com/golang/go/issues/26728, July 2018. [Online; accessed on 03/04/2020].

[52] L. Mastrangelo, M. Hauswirth, and N. Nystrom. Casting about in the dark: An empirical study of cast operations in java programs. *Proc. ACM Program. Lang.*, 3(OOPSLA):158:1–158:31, Oct. 2019.

[53] L. Mastrangelo, L. Ponzanelli, A. Mocci, M. Lanza, M. Hauswirth, and N. Nystrom. Use at your own risk: The java unsafe api in the wild. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 695–710, New York, NY, USA, 2015. ACM.

[54] D. Mazinanian, A. Ketkar, N. Tsantalis, and D. Dig. Understanding the use of lambda expressions in java. *Proc. ACM Program. Lang.*, 1(OOPSLA), Oct. 2017.

[55] M. McHugh. Interrater reliability: The kappa statistic. *Biochemia medica : časopis Hrvatskoga društva medicinskih biokemičara / HDMB*, 22:276–82, 10 2012.

[56] V. Mihailenco. Use unsafe for bytes to string conversion. https://github.com/go-redis/redis/pull/138, Jul 2015. (Accessed on 10/30/2020).

[57] V. Mihailenco. Use unsafe to reduce number of allocations. https://github.com/go-pg/pg/pull/272, Feb 2016. (Accessed on 10/30/2020).

[58] V. Mihailenco. Use unsafe stringtobytes to speedup stringcmd.bytes by vmihailenco. https://github.com/go-redis/redis/pull/1106, Jul 2019. (Accessed on 10/30/2020).

[59] M. Nagappan, R. Robbes, Y. Kamei, E. Tanter, S. McIntosh, A. Mockus, and A. E. Hassan. An empirical study of goto in c code from github repositories. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 404–414, New York, NY, USA, 2015. ACM.

[60] C. Project. Cilium. https://cilium.io/. [Online; accessed on 01/03/2020].

[61] V. Romanov. Usafe use of unsafe that leads to data corruption. https://github.com/francoispqt/gojay/issues/3, Apr 2018. [Online; accessed on 03/04/2020].

[62] T. Sandall. Remove use of unsafe.pointer for string hashing by tsandall. https://github.com/open-policy-agent/opa/pull/602, Feb 2018. [Online; accessed on 03/04/2020].

[63] W. Schulze. No unsafe by awalterschulze · pull request #343 · gogo/protobuf. https://github.com/gogo/protobuf/pull/343, Oct 2017. [Online; accessed on 01/03/2020].

[64] B. Siegert. unsafe.sizeof now returns uintptr · issue #3 · lxn/walk. https://github.com/lxn/walk/issues/3, Jun 2011. [Online; accessed on 01/03/2020].

[65] D. Smirnov. Known issue: unsafe (marshaler and unmarshaler) test failures on big endian architectures. https://github.com/gogo/protobuf/issues/195, Aug 2016. [Online; accessed on 01/03/2020].

[66] C. Stockton. [security] incorrect unsafe usage potentially exposes prior request parameters. https://github.com/nuclio/nuclio/issues/277, Oct 2017. [Online; accessed on 03/04/2020].

[67] I. L. Taylor. proposal: Go 2: only give special properties to unsafe.pointer if package imports unsafe. https://github.com/golang/go/issues/26070. [Online; accessed on 01/08/2019].

[68] P. Wang, G. R. Bai, and K. T. Stolee. Exploring regular expression evolution. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 502–513. IEEE, 2019.
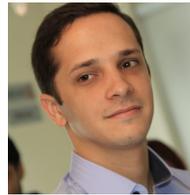
Diego Elias Costa is a postdoctoral researcher in the Department of Computer Science and Software Engineering at Concordia University. He received his PhD in Computer Science from Heidelberg University, Germany. His research interests cover a wide range of software engineering and performance engineering related topics, including mining software repositories, software ecosystems, and performance testing. You can find more about him at http://das.encs.concordia.ca/members/diego-costa/.

Suhaib Mujahid is a Ph.D. student in the Department of Computer Science and Software Engineering at Concordia University. He received his master's in Software Engineering from Concordia University (Canada) in 2017, where his work focused on detection and mitigation of permission-related issues facing wearable app developers. He did his Bachelors in Information Systems at Palestine Polytechnic University. His research interests include wearable applications, software quality assurance, mining software repositories and empirical software engineering. You can find more about him at http://users.encs.concordia.ca/smujahi.

Rabe Abdalkareem received his PhD in Computer Science and Software Engineering from Concordia University. His research investigates how the adoption of crowdsourced knowledge affects software development and maintenance. Abdalkareem received his master's in applied computer science from Concordia University. His work has been published at premier venues such as FSE, ICSME, and MobileSoft, as well as in major journals such as TSE, IEEE Software, EMSE and IST. You can find more about him at http://users.encs.concordia.ca/rababdu.

Emad Shihab is an associate professor in the Department of Computer Science and Software Engineering at Concordia University. He received his PhD from Queens University. Dr. Shihabs research interests are in Software Quality Assurance, Mining Software Repositories, Technical Debt, Mobile Applications and Software Architecture. He worked as a software research intern at Research In Motion in Waterloo, Ontario and Microsoft Research in Redmond, Washington. Dr. Shihab is a member of the IEEE and ACM. More information can be found at http://das.encs.concordia.ca.